# IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

## A PATENT APPLICATION FOR

## TRACE INDEXING VIA TRACE END ADDRESSES

| | |
|---|---|
| INVENTORS: | Stephan Jourdan<br>Lihu Rappoport<br>Ronny Ronen<br>Adi Yoaz<br>of Intel Corporation |
| PREPARED BY: | Robert L. Hails Jr.<br>KENYON & KENYON<br>1500 K Street, N.W.<br>Washington, DC 20005<br>(202) 220-4200 |

# TRACE INDEXING VIA TRACE END ADDRESSES

## BACKGROUND

The present invention provides a new trace assembly paradigm for processing circuits.

FIG. 1 is a block diagram illustrating the process of program execution in a conventional processor. Program execution may include three stages: front end 110, execution 120 and memory 130. The front-end stage 110 performs instruction pre-processing. Front end processing is designed with the goal of supplying valid decoded instructions to an execution core with low latency and high bandwidth. Front-end processing can include instruction prediction, decoding and renaming.

As the name implies, the execution stage 120 performs instruction execution. The execution stage 120 typically communicates with a memory 130 to operate upon data stored therein.

The front end stage 110 may include a trace cache (not shown) to reduce the latency of instruction decoding and to increase front end bandwidth. A trace cache is a circuit that assembles sequences of dynamically executed instructions into logical units called "traces." The program instructions may have been assembled into a trace from non-contiguous regions of an external memory space but, when they are assembled in a trace, the instructions appear in program order. Typically, a trace may begin with an instruction of any type. The trace may end when one of number of predetermined trace end conditions occurs, such as a trace size limit, a maximum number of conditional branches occurs or an indirect branch or return instruction occurs.

Prior art traces are defined by an architecture having a single entry point but possibly many exit points. This architecture, however, causes traces to exhibit instruction redundancy. Consider, by way of example, the following code sequence:

```
If (cond)
        A
B
```

This simple code sequence produces two possible traces: 1) B and 2) AB. When assembled in a trace cache, each trace may be stored independently of the other. Thus, the B instruction would be stored twice in the trace cache. This redundancy lowers system performance. Further, because traces may start on any instruction, the B instruction also may be recorded in multiple traces due to instruction alignment discrepancies that may occur. This instruction redundancy limits the bandwidth of front-end processing.

Accordingly, there is a need in the art for a front-end stage that reduces instruction redundancy in traces.

## **BRIEF DESCRIPTION OF THE DRAWINGS**

FIG. 1 is a block diagram illustrating functional processing in program execution.

FIG. 2 is a block diagram of a front-end stage 200 according to an embodiment of the present invention.

FIG. 3 is a flow diagram illustration operation 1000 of the XFU 260 according to an embodiment of the invention.

FIGS. 4(a)-4(c) illustrate construction of extended blocks according to embodiments of the present invention.

FIG. 5 illustrates construction of an extended block according to an embodiment of the present invention.

FIG. 6 illustrates construction of a complex extended block according to a embodiment of the present invention.

## **DETAILED DESCRIPTION**

Embodiments of the present invention assemble a new type of traces, called "extended blocks" herein, according to an architecture that permits several entry points but only a single exit point. These extended blocks may be indexed based upon the address of the last instruction therein. The extended block architecture provides several advantages over prior architectures:

- instruction redundancies can be eliminated,
- multiple entry points are permitted,
- extended blocks may be extended dynamically, and
- basic blocks may be shared among various extended blocks.

FIG. 2 is a block diagram of a front-end stage 200 according to an embodiment of the present invention. The front end 200 may include an instruction cache 210 and an extended block cache ("XBC") 220. The instruction cache 210 may be based on any number of known architectures for front-end systems. Typically, they include an instruction memory 230, a branch predictor 240 and an instruction decoder 250. Program instructions may be stored in the cache memory 230 and indexed by an instruction pointer. Instructions may be retrieved from the cache memory 230, decoded by the instruction decoder 250 and passed to the execution unit (not shown). The branch predictor 240 may assist in the selection of instructions to be retrieved from the cache memory 230 for execution. As is known, instructions may be indexed by an address, called an "instruction pointer" or "IP."

According to an embodiment, an XBC 220 may include a fill unit ("XFU") 260, a block predictor ("XBTB") 270 and a block cache 280. The XFU 260 may build the extended blocks. The block cache 280 may store the extended blocks. The XBTB 270 may predict which extended blocks, if any, are likely to be executed and may cause the block cache to furnish any predicted blocks to the execution unit. The XBTB 270 may store masks associated with each of the extended blocks stored by the block cache 280, indexed by the IP of the terminal instruction of the extended blocks.

The XBC 220 may receive decoded instructions from the instruction cache 210. The XBC 220 also may pass decoded instructions to the execution unit (not shown). A selector 290 may select which front-end source, either the instruction cache 210 or the XBC 220, will supply instructions to the execution unit. In an embodiment, the block cache 280 may control the selector 290.

As discussed, the block cache 280 may be a memory that stores extended blocks. According to an embodiment, the extended blocks may be multiple-entry, single-exit traces. An extended block may include a sequence of program instructions. It may terminate in a conditional branch, an indirect branch or based upon a predetermined termination condition such as a size limit. Again, the block cache 280 may index the extended blocks based upon an IP of the terminal instruction in the block.

Extended blocks are useful because, whenever program flow enters the extended block, the flow necessarily progresses to the terminal instruction in the extended block. An extended block may contain any conditional or indirect branches only as a terminal instruction. Thus, the extended block may have multiple entry points. According to an embodiment, an unconditional branch need not terminate an extended block.

According to an embodiment, a hit/miss indication from the block cache 280 may control the selector 290.

FIG. 3 is a flow diagram illustrating operation 1000 of the XBC 220 according to an embodiment of the invention. Operation may begin when the XBC 220 determines whether an IP of a terminal instruction from an extended block may be predicted (Stage 1010). If so, the XBC 200 performs such a prediction (Stage 1020). The prediction may be made by the XBTB 220. Based on the IP of the predicted terminal instruction, the block cache 280 may indicate a "hit" or a "miss" (Stage 1030). A hit indicates that the block cache 280 stores an extended block that terminates at the predicted IP. In this case, the XFU 260 may cause an extended block to be retrieved from the block cache 280 and forwarded to the execution units (Stage 1040). Thereafter, the process may return to Stage 1010 and repeat.

If the predicted IP does not hit the block cache 280 or if an IP of a terminal instruction could not be predicted at Stage 1010, the XFU 260 may build a new extended block. Decoded instructions from the instruction cache may be forwarded to the execution unit (Stage 1050). The XFU 260 also may receive the retrieved instructions from the instruction cache system 210. It stores instructions in a new block until it reaches a terminal condition, such as a conditional or implicit branch or a size limitation

(Stage 1060). Having assembled a new block, the XFU 260 may determine how to store it in the block cache 280.

The XFU 260 may determine whether the terminal IP of the new block hits the block cache (Stage 1070). If not, then the XFU 260 simply may cause the new block to be stored in the block cache 280 (Stage 1080).

If the IP hits the block cache, then the XFU 260 may compare the contents of the new block with the contents of an older block stored in the block cache that generated the hit. The XFU 260 may determine whether the contents of the new block are subsumed entirely within the old block (Stage 1090). If so, the XFU 260 need not store the new block in the block cache 280 because it is present already in the old block. If not, the XFU 260 may determine whether the contents of the old block are subsumed within the new block (Stage 1100). If so, the XFU 260 may write the new block over the old block in the cache (Stage 1110). This has the effect of extending the old block to include the new block.

If neither of the above conditions is met, then the new block and the old block may be only partially co-extensive. There are several alternatives for this case. In a first embodiment, the XFU 260 may store the non-overlapping portion of the new block in the block cache 280 as a separate extended block (Stage 1120). Alternatively, the XFU 260 may create a complex extended block from the new block and the old block (Stage 1130). These are discussed in greater detail below.

Once the new block is stored in the block cache 280, at the conclusion of Stages 1110, 1120 or 1130, the XBTB may be updated to reflect the contents of the block cache 280 (Stage 1140). Thereafter, operation may return to Stage 1010 for a subsequent iteration.

Sub B1

FIGS. 4(a)-4(c) schematically illustrate the different scenarios that may occur if the IP pointer of the ~~new extended~~ block $XB_{new}$ matches that of an extended block stored previously within the block cache ~~280 ($XB_{old}$).~~ In FIG. 4(a), the older extended block $XB_{old}$ is co-extensive with the new extended block $XB_{new}$ but is ~~longer.~~ In this case, the new extended block $XB_{new}$ should not be stored separately within the block cache ~~280,~~

the older extended block may be used instead. In FIG. 4(a), the older extended block $XB_{old}$ is co-extensive with a portion of the new extended $XB_{new}$ but $XB_{new}$ is longer. In this case, the older extended block may be extended to include the additional instructions found in $XB_{new}$.

In a third case, shown in FIG. 4(c), only a portion of $XB_{new}$ and $XB_{old}$ are co-extensive. In this case, a "suffix" of each extended block coincides but the two extended blocks have different "prefixes." In a first embodiment, the XFU 260 may store a prefix of the new extended block as an extended block all its own. The prefix may end in an unconditional jump pointing to an appropriate location in the older extended block. This solution is shown in FIG. 5.

Alternatively, the XFU 260 may assemble a single, complex extended block merging the two extended blocks. In this embodiment, the XFU 260 may extend the older extended block by adding the prefix of the new extended block to a head of the older extended block. The prefix of the new extended block may conclude in an unconditional jump pointing to the common suffix. This solution is shown in FIG. 6. This embodiment creates a single, longer extended block instead of two short extended blocks and thereby contributes to increased bandwidth. In such an embodiment, the XFU 260 may generate a mask in the block cache 280 that is associated with the complex extended block.

According to an embodiment, the XBTB 270 may predict extended blocks to be used.

Returning to FIG. 2, the XBTB 270 may predict extended blocks for use during program execution. The XBTB 270 may store information for each of the extended blocks stored in the block cache 280. In an embodiment, the XBTB 270 may store masks for each block cache. These masks may identify whether a corresponding extended block is a complex or non-complex block (compare FIGS. 4(a) and 4(b) with FIG. 6). For non-complex extended blocks, the XBTB 270 may store a mask identifying the length of the extended blocks. For complex extended blocks, the XBTB 270 may store a mask that distinguishes multiple prefixes from each other. Thus, while the prefixes may be linearly appended to each other as shown in FIG. 6, a mask as stored in the XBTB permits the

XBTB to determine the position of an entry point to an extended block based on an instruction's IP.

As described above, an XBTB 270 may store a mapping among blocks. As noted, the XBTB 270 may identify the terminal IP of each block and may store masks for each extended block. The XBTB 270 also may store a mapping identifying links from one XBTB to another. For example, if a first extended block ended in a conditional branch, program execution may reach a second extended block by following one of the directions of the terminal branch and execution may reach a third extended block by following the other direction. The XBTB 270 may store a mapping of both blocks. In this embodiment, the XFU 260 may interrogate the XBTB 270 with an IP. By way of response, the XBC 220 may respond with a hit or miss indication and, if the response is a hit, may identify one or more extended blocks in the block cache to which the IP may refer. In this embodiment, the XBC 220 may determine whether a terminal IP may be predicted and whether the predicted address hits the cache (Stages 1010, 1030).

According to an embodiment, XBC bandwidth may be improved by using conditional branch promotion for terminal instructions. As is known, conditional branch promotion permits a conditional branch to be treated as an unconditional branch if it is found that the branch is nearly monotonic. For example, if during operation, it is determined that program execution at a particular branch tends to follow one of the branches 99% of the time, the branch may be treated as an unconditional jump. In this case, two extended blocks may be joined as a single extended block. This embodiment further contributes to XBC bandwidth.

According to another embodiment, XBC latency may be minimized by having the block cache 280 store extended block instructions in decoded form. Thus, the instructions may be stored as microinstructions (commonly, "uops").

Several embodiments of the present invention are specifically illustrated and described herein. However, it will be appreciated that modifications and variations of the present invention are covered by the above teachings and within the purview of the appended claims without departing from the spirit and intended scope of the invention.